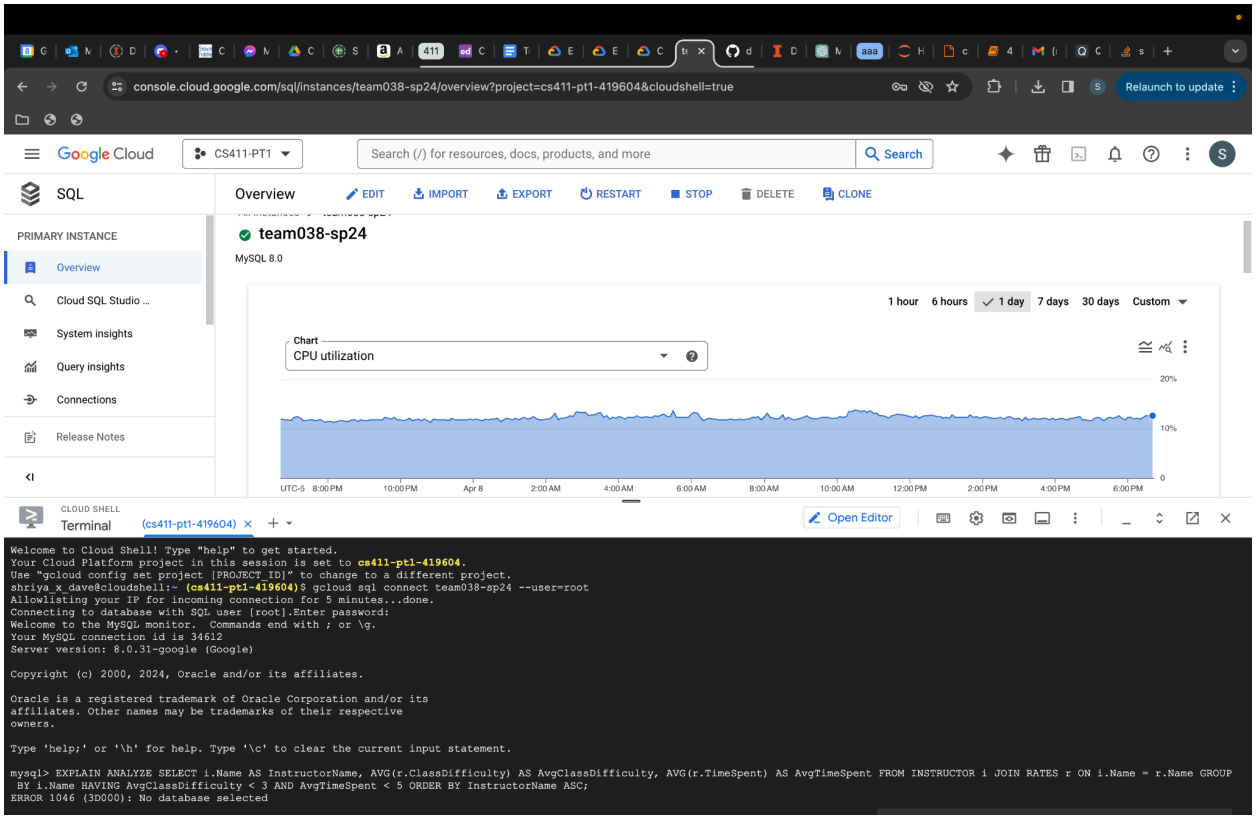
Database Implementation

Connection Proof



DDL

```
CREATE TABLE User (
    NetId VARCHAR(20) PRIMARY KEY,
    FirstName VARCHAR(255),
    LastName VARCHAR(255),
    Email VARCHAR(255),
    Year VARCHAR(20)
);
CREATE TABLE Course (
    CourseId VARCHAR(10) PRIMARY KEY,
    CRN INT,
    Credits INT,
    GenEdCriteria VARCHAR(255),
    Location VARCHAR(255),
    MeetingTime VARCHAR(10),
    AverageGPA FLOAT
);
CREATE TABLE Instructor (
```

```
    Name VARCHAR(255) PRIMARY KEY,
    Email VARCHAR(255),
    Title VARCHAR(255),
    FOREIGN KEY (Title) REFERENCES Department(Title)
);
CREATE TABLE Department (
    Title VARCHAR(255) PRIMARY KEY,
    Building VARCHAR(255)
);
CREATE TABLE Research (
    ResearchName VARCHAR(255) PRIMARY KEY,
    Pay FLOAT,
    Field VARCHAR(255),
    Program VARCHAR(255)
);
CREATE TABLE Enrolls (
    NetID VARCHAR(255),
    CRN INT,
    PRIMARY KEY (NetID, CRN),
    FOREIGN KEY (NetID) REFERENCES User(NetId),
    FOREIGN KEY (CRN) REFERENCES Course(CRN)
);
CREATE TABLE Rates (
    NetID VARCHAR(255),
    Name VARCHAR(255),
    Rating INT,
    ClassDifficulty FLOAT,
    TimeSpent INT,
    PRIMARY KEY (NetID, Name),
    FOREIGN KEY (NetID) REFERENCES User(NetId),
    FOREIGN KEY (Name) REFERENCES Instructor(Name)
);
CREATE TABLE Teaches (
    CRN INT,
    Name VARCHAR(255),
    PRIMARY KEY (CRN, Name),
    FOREIGN KEY (CRN) REFERENCES Course(CRN),
    FOREIGN KEY (Name) REFERENCES Instructor(Name)
);
CREATE TABLE Offers (
```

```
    CRN INT,
    Title VARCHAR(255),
    PRIMARY KEY (CRN, Title),
    FOREIGN KEY (CRN) REFERENCES Course(CRN),
    FOREIGN KEY (Title) REFERENCES Department(Title)
);
CREATE TABLE Contributes (
    ResearchName VARCHAR(255),
    NetID VARCHAR(255),
    Role VARCHAR(255),
    PRIMARY KEY (ResearchName, NetID),
    FOREIGN KEY (ResearchName) REFERENCES Research(ResearchName),
    FOREIGN KEY (NetID) REFERENCES User(NetId)
);
CREATE TABLE Leads (
    ResearchName VARCHAR(255),
    Name VARCHAR(255),
    PRIMARY KEY (ResearchName, Name),
    FOREIGN KEY (ResearchName) REFERENCES Research(ResearchName),
    FOREIGN KEY (Name) REFERENCES Instructor(Name)
);
```

Proof of 1000 Rows in 3 Tables

The three tables we have with over 1000 rows are User, Instructor, and Course

```
mysql> SELECT COUNT(*) FROM USER;
+----------+
| COUNT(*) |
+----------+
|     1483 |
+----------+
1 row in set (0.00 sec)
```

```
mysql> SELECT COUNT(*) FROM INSTRUCTOR;
+----------+
| COUNT(*) |
+----------+
|     9318 |
+----------+
1 row in set (0.00 sec)
```

```
mysql> SELECT COUNT(*) FROM COURSE;
+----------+
| COUNT(*) |
+----------+
|     1748 |
+----------+
1 row in set (0.01 sec)
```

Advanced SQL Queries:

Query1: Instructors with Low Time Spent, Class Difficulty and Rating

```
SELECT
   i.Name AS InstructorName,
   MAX(r.ClassDifficulty) AS MaxClassDifficulty,
   MAX(r.TimeSpent) AS MaxTimeSpent,
   MAX(r.Rating) AS MaxRating
FROM
   INSTRUCTOR i
JOIN
   RATES r ON i.Name = r.Name
GROUP BY
   i.Name
HAVING
   MAX(r.TimeSpent) < 5
   AND MAX(r.ClassDifficulty) < 3
   AND MAX(r.Rating) < 3
ORDER BY
   InstructorName ASC
LIMIT 15;
```

InstructorName	MaxClassDifficulty	MaxTimeSpent	MaxRating
Beardsley, Erik L	1	4	1
Brown, Stephanie A	2	1	1
Bui, Long B	1	4	2
Deloff, Christine A	2	1	1
Denardo, Gregory F	1	3	1
Dorantes, Lizette	1	2	2
Hagen, Wyatt A	2	2	2
Kutz, Elaina D	2	2	1
Liu, Yao-Min	1	3	1
Madak-Erdogan, Zeynep	1	4	1
Maeng, Junghwan	1	3	2
Mitchell, Rebecca A	1	1	2
Nafziger, Emerson D	2	4	1
Name	0	0	0
O'Toole, Kathryn E	1	4	1

15 rows in set (0.01 sec)

Query 2: Average Instructor Rating

```
SELECT i.Name AS InstructorName, c.CourseID, i.Title,  AVG(r.Rating) AS
AverageInstructorRating FROM INSTRUCTOR i NATURAL JOIN RATES r JOIN TEACHES t
ON i.Name = t.Name JOIN COURSE c
```

ON t.CRN = c.CRN WHERE c.CourseID LIKE "S%" OR i.Title LIKE "S%" GROUP BY i.Name, c.CourseID ORDER BY AverageInstructorRating DESC LIMIT 15;

InstructorName	CourseID	Title	AverageInstructorRating
Dominguez, Virginia R	STAT 107	ANTH	5.0000
Bin Khidzer, Mohammad Kha	PHYS 214	SOC	4.0000
Studamire, Dante R	STAT 107	EDUC	4.0000
Williams, Amy K	STAT 207	GLBL	4.0000
Simon, Pedro V	SOC 100	ECON	4.0000
Bobitt, Julie L	SPAN 232	IHLT	3.0000
Cantet, Maria N	SPAN 248	ECON	3.0000
Clarke, Caitlin L	ANTH 101	SOC	3.0000
Gay, Lucy C	STAT 107	GS	3.0000
Keenan, Kimberly A	ECON 103	SOCW	3.0000
Xue, Fei	REL 110	STAT	3.0000
Mette, Alan T	SPAN 248	ARTF	3.0000
Moqadam, Mahbubeh	HIST 279	SOC	3.0000
Moqadam, Mahbubeh	GWS 275	SOC	3.0000
Simons, Claire L	STAT 207	CHEM	3.0000

Query 3: Departments with Either High Class Difficulty, Time Spent or Low Rating

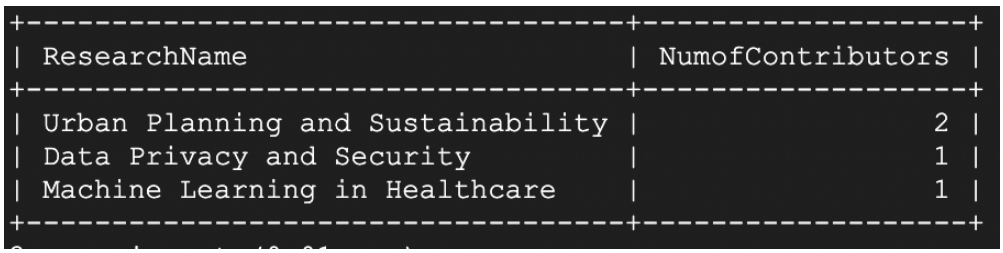
SELECT d.Title as DepartmentTitle, AVG(r.ClassDifficulty) AS AvgClassDifficulty, AVG(r.TimeSpent) AS AvgTimeSpent, AVG(r.Rating) AS AvgRating
FROM Rates r NATURAL JOIN Instructor i JOIN Department d ON i.Title = d.Title
GROUP BY d.Title
HAVING AvgClassDifficulty > 5 OR AvgTimeSpent > 10 OR AvgRating < 3
ORDER BY DepartmentTitle ASC
LIMIT 15;

DepartmentTitle	AvgClassDifficulty	AvgTimeSpent	AvgRating
AAS	5.3333	9.8333	3.0000
ABE	4.2000	12.2000	3.8000
ACCY	6.0976	9.7073	2.7317
ACE	5.7407	9.9630	3.2222
ACES	6.2000	11.9000	2.7000
ADV	5.7647	9.7059	3.0588
AE	5.6667	11.9167	3.5833
AGCM	5.6667	8.0000	3.3333
AGED	6.3333	10.3333	1.0000
AHS	5.2000	11.0000	3.2000
AIS	5.5000	15.5000	2.5000
ANSC	4.1429	11.4286	3.0000
ANTH	5.4444	10.7222	2.8889
ARCH	5.3793	11.8276	2.8621
ART	3.3077	10.4615	2.7692

Query 4: Top Research Projects by Contributor Count

SELECT r.ResearchName as ResearchName, COUNT(*) AS NumofContributors
FROM CONTRIBUTES c NATURAL JOIN RESEARCH r
WHERE c.Role LIKE "Lab%"

```
 GROUP BY r.ResearchName HAVING NumOfContributors > 0
ORDER BY NumOfContributors
DESC LIMIT 15;
```



ResearchName	NumofContributors
Urban Planning and Sustainability	2
Data Privacy and Security	1
Machine Learning in Healthcare	1

This query produced only 3 rows instead of 15 due to the primary key constraint on the Research entity, which is uniquely identified by ResearchName and the where clause. When multiple researchers name their projects the same thing, all but one of these projects are excluded from the output. As a result, only unique research projects are counted and displayed in the results.

Indexing

Query 1– Instructors with Low Time Spent, Class Difficulty and Teacher Rating

Before Indexing

```
-> Limit: 15 row(s)  (actual time=8.140..8.142 rows=15 loops=1)
  -> Sort: i.`Name`  (actual time=8.138..8.139 rows=15 loops=1)
      -> Filter: ((max(r.TimeSpent) < 5) and (max(r.ClassDifficulty) < 3) and (max(r.Rating) < 3))  (actual time=7.786..8.099 rows=24 loops=1)
          -> Table scan on <temporary>  (actual time=7.769..7.992 rows=1384 loops=1)
              -> Aggregate using temporary table  (actual time=7.765..7.765 rows=1384 loops=1)
                  -> Nested loop inner join  (cost=677.70 rows=1501) (actual time=0.141..5.695 rows=1504 loops=1)
                      -> Table scan on r  (cost=152.35 rows=1501) (actual time=0.111..0.943 rows=1504 loops=1)
                      -> Single-row covering index lookup on i using PRIMARY (Name=r.`Name`)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=1504)
```

After Indexing on Class Difficulty:

```
| -> Limit: 15 row(s)  (actual time=7.264..7.267 rows=15 loops=1)
    -> Sort: i.`Name`  (actual time=7.263..7.264 rows=15 loops=1)
        -> Filter: ((max(r.TimeSpent) < 5) and (max(r.ClassDifficulty) < 3) and (max(r.Rating) < 3))  (actual time=6.767..7.212 rows=24 loops=1)
            -> Table scan on <temporary>  (actual time=6.748..7.093 rows=1384 loops=1)
                -> Aggregate using temporary table  (actual time=6.744..6.744 rows=1384 loops=1)
                    -> Nested loop inner join  (cost=677.70 rows=1501) (actual time=0.180..5.004 rows=1504 loops=1)
                        -> Table scan on r  (cost=152.35 rows=1501) (actual time=0.097..0.751 rows=1504 loops=1)
                        -> Single-row covering index lookup on i using PRIMARY (Name=r.`Name`)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=1504)
 |
```

After Indexing on Time Spent:

```
| -> Limit: 15 row(s)  (actual time=7.753..7.755 rows=15 loops=1)
    -> Sort: i.`Name`  (actual time=7.752..7.753 rows=15 loops=1)
        -> Filter: ((max(r.TimeSpent) < 5) and (max(r.ClassDifficulty) < 3) and (max(r.Rating) < 3))  (actual time=7.397..7.722 rows=24 loops=1)
            -> Table scan on <temporary>  (actual time=7.382..7.617 rows=1384 loops=1)
                -> Aggregate using temporary table  (actual time=7.379..7.379 rows=1384 loops=1)
                    -> Nested loop inner join  (cost=677.70 rows=1501) (actual time=0.090..5.376 rows=1504 loops=1)
                        -> Table scan on r  (cost=152.35 rows=1501) (actual time=0.067..0.884 rows=1504 loops=1)
                        -> Single-row covering index lookup on i using PRIMARY (Name=r.`Name`)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=1504)
 |
```

After Indexing on Rating:

```
| -> Limit: 15 row(s)  (actual time=6.245..6.248 rows=15 loops=1)
    -> Sort: i.`Name`  (actual time=6.244..6.246 rows=15 loops=1)
        -> Filter: ((max(r.TimeSpent) < 5) and (max(r.ClassDifficulty) < 3) and (max(r.Rating) < 3))  (actual time=5.825..6.214 rows=24 loops=1)
            -> Table scan on <temporary>  (actual time=5.810..6.082 rows=1384 loops=1)
                -> Aggregate using temporary table  (actual time=5.807..5.807 rows=1384 loops=1)
                    -> Nested loop inner join  (cost=677.70 rows=1501) (actual time=0.082..4.265 rows=1504 loops=1)
                        -> Table scan on r  (cost=152.35 rows=1501) (actual time=0.061..0.675 rows=1504 loops=1)
                        -> Single-row covering index lookup on i using PRIMARY (Name=r.`Name`)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1504)
 |
```

Justification:
Throughout the different parameters that are indexed for query 1, there is no change in cost across all of them. This is because all of these attributes are in the HAVING clause. Indexing columns used in aggregate functions and subsequently in the HAVING clause is less likely to affect the cost as indicated by the query planner because the HAVING clause filters the results that are not indexed, which means the database can't leverage the benefits of index scanning. However, there are significant differences in time. Indexing on ClassDifficulty or TimeSpent increased the times of the limit operation, sort operation, filter operation, both table scans, aggregation, and nested loop inner join with indexing on TimeSpent being slightly lower than ClassDifficulty, but still taking longer than pre-indexing. On the other hand, indexing on Rating yielded lower times in all of the categories stated above in comparison to the pre-indexing times. These results that reveal changes in time might be due to the time needed to manage the index, so it could be variable. Even though the cost associated with the nested loop inner join and table scans remain unchanged, the reduction in the actual time taken for the overall query indicated that the indexing did have a positive effect. Indexing on Rating was the best option since it's used in the join between Instructor and Rates. In comparison to ClassDifficulty and TimeSpent, Rating is better because of its higher selectivity since rating values are likely to vary more which

means that they would be able to narrow down results more effectively. An index on this attribute could speed up the join operation quickly by being able to locate the corresponding entries on both sides.

Query 2 – Average Instructor Rating:

Before Indexing:

```
-------------------------------------------------------------------------------+
 Limit: 15 row(s)  (actual time=6.745..6.749 rows=15 loops=1)
-> Sort: AverageInstructorRating DESC, limit input to 15 row(s) per chunk  (actual time=6.744..6.746 rows=15 loops=1)
    -> Table scan on <temporary>  (actual time=6.675..6.682 rows=23 loops=1)
        -> Aggregate using temporary table  (actual time=6.670..6.670 rows=23 loops=1)
            -> Nested loop inner join  (cost=962.41 rows=878) (actual time=0.564..6.535 rows=26 loops=1)
                -> Nested loop inner join  (cost=672.36 rows=878) (actual time=0.257..5.845 rows=141 loops=1)
                    -> Nested loop inner join  (cost=365.05 rows=809) (actual time=0.097..2.136 rows=809 loops=1)
                        -> Covering index scan on t using Name  (cost=81.90 rows=809) (actual time=0.073..0.383 rows=809 loops=1)
                        -> Single-row index lookup on c using PRIMARY (CRN=t.CRN)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=809)
                    -> Index lookup on r using Name (Name=t.`Name`)  (cost=0.27 rows=1) (actual time=0.004..0.004 rows=0 loops=809)
                -> Filter: ((c.CourseID like 'S%') or (i.Title like 'S%'))  (cost=0.23 rows=1) (actual time=0.005..0.005 rows=0 loops=141)
                    -> Single-row index lookup on i using PRIMARY (Name=t.`Name`)  (cost=0.23 rows=1) (actual time=0.004..0.004 rows=1 loops=141)
```

After Indexing Rating:

```
-------------------------------------------------------------------------------+
Limit: 15 row(s)  (actual time=5.041..5.043 rows=15 loops=1)
> Sort: AverageInstructorRating DESC, limit input to 15 row(s) per chunk  (actual time=5.039..5.041 rows=15 loops=1)
    -> Table scan on <temporary>  (actual time=5.005..5.009 rows=23 loops=1)
        -> Aggregate using temporary table  (actual time=5.002..5.002 rows=23 loops=1)
            -> Nested loop inner join  (cost=962.41 rows=878) (actual time=0.554..4.928 rows=26 loops=1)
                -> Nested loop inner join  (cost=672.36 rows=878) (actual time=0.249..4.397 rows=141 loops=1)
                    -> Nested loop inner join  (cost=365.05 rows=809) (actual time=0.085..1.632 rows=809 loops=1)
                        -> Covering index scan on t using Name  (cost=81.90 rows=809) (actual time=0.066..0.278 rows=809 loops=1)
                        -> Single-row index lookup on c using PRIMARY (CRN=t.CRN)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=809)
                    -> Index lookup on r using Name (Name=t.`Name`)  (cost=0.27 rows=1) (actual time=0.003..0.003 rows=0 loops=809)
                -> Filter: ((c.CourseID like 'S%') or (i.Title like 'S%'))  (cost=0.23 rows=1) (actual time=0.004..0.004 rows=0 loops=141)
                    -> Single-row index lookup on i using PRIMARY (Name=t.`Name`)  (cost=0.23 rows=1) (actual time=0.003..0.003 rows=1 loops=141)
```

After Indexing CourseID:

```
-------------------------------------------------------------------------------+
 Limit: 15 row(s)  (actual time=4.922..4.924 rows=15 loops=1)
-> Sort: AverageInstructorRating DESC, limit input to 15 row(s) per chunk  (actual time=4.921..4.922 rows=15 loops=1)
    -> Table scan on <temporary>  (actual time=4.861..4.867 rows=23 loops=1)
        -> Aggregate using temporary table  (actual time=4.857..4.857 rows=23 loops=1)
            -> Nested loop inner join  (cost=962.41 rows=878) (actual time=0.444..4.778 rows=26 loops=1)
                -> Nested loop inner join  (cost=672.36 rows=878) (actual time=0.203..4.304 rows=141 loops=1)
                    -> Nested loop inner join  (cost=365.05 rows=809) (actual time=0.064..1.521 rows=809 loops=1)
                        -> Covering index scan on t using Name  (cost=81.90 rows=809) (actual time=0.050..0.273 rows=809 loops=1)
                        -> Single-row index lookup on c using PRIMARY (CRN=t.CRN)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=809)
                    -> Index lookup on r using Name (Name=t.`Name`)  (cost=0.27 rows=1) (actual time=0.003..0.003 rows=0 loops=809)
                -> Filter: ((c.CourseID like 'S%') or (i.Title like 'S%'))  (cost=0.23 rows=1) (actual time=0.003..0.003 rows=0 loops=141)
                    -> Single-row index lookup on i using PRIMARY (Name=t.`Name`)  (cost=0.23 rows=1) (actual time=0.003..0.003 rows=1 loops=141)
```

After Indexing Title:

```
-------------------------------------------------------------------------------+
> Limit: 15 row(s)  (actual time=5.008..5.011 rows=15 loops=1)
 -> Sort: AverageInstructorRating DESC, limit input to 15 row(s) per chunk  (actual time=5.007..5.009 rows=15 loops=1)
     -> Table scan on <temporary>  (actual time=4.972..4.977 rows=23 loops=1)
         -> Aggregate using temporary table  (actual time=4.969..4.969 rows=23 loops=1)
             -> Nested loop inner join  (cost=962.41 rows=878) (actual time=0.506..4.893 rows=26 loops=1)
                 -> Nested loop inner join  (cost=672.36 rows=878) (actual time=0.268..4.435 rows=141 loops=1)
                     -> Nested loop inner join  (cost=365.05 rows=809) (actual time=0.079..1.640 rows=809 loops=1)
                         -> Covering index scan on t using Name  (cost=81.90 rows=809) (actual time=0.063..0.281 rows=809 loops=1)
                         -> Single-row index lookup on c using PRIMARY (CRN=t.CRN)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=809)
                     -> Index lookup on r using Name (Name=t.`Name`)  (cost=0.27 rows=1) (actual time=0.003..0.003 rows=0 loops=809)
                 -> Filter: ((c.CourseID like 'S%') or (i.Title like 'S%'))  (cost=0.23 rows=1) (actual time=0.003..0.003 rows=0 loops=141)
                     -> Single-row index lookup on i using PRIMARY (Name=t.`Name`)  (cost=0.23 rows=1) (actual time=0.003..0.003 rows=1 loops=141)
```

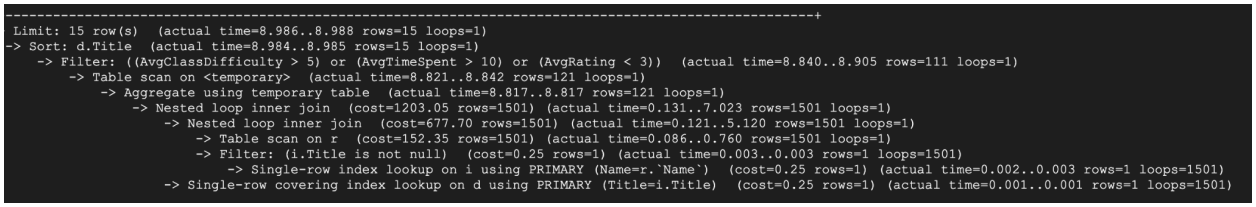
Justification:

After indexing on Query 2, the cost does not really change for any of the attributes once again. However, the query efficiency does improve based on time. Since the query is calculating average instructor rating, indexing on Rating allows the database to quickly access and compute the average without having to go through the entire Rates table. This is especially useful for aggregation operations since the index significantly reduces the amount of data that needs to be processed. Rating also provides a direct path to the data needed for the AVG function, which

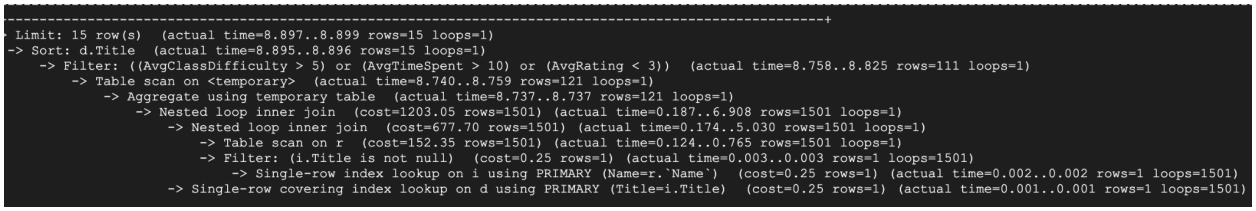
benefits the join operation by helping indirectly optimize the ordering of the results. Indexing on CourseId also reduces time by almost 2 seconds since it is used in a join operation and speeds up the operation. Finally, indexing on Course Title also improved efficiency based on the time but not very significantly.

Query 3 – <u>Departments with Above-Average Class Difficulty Ratings</u>

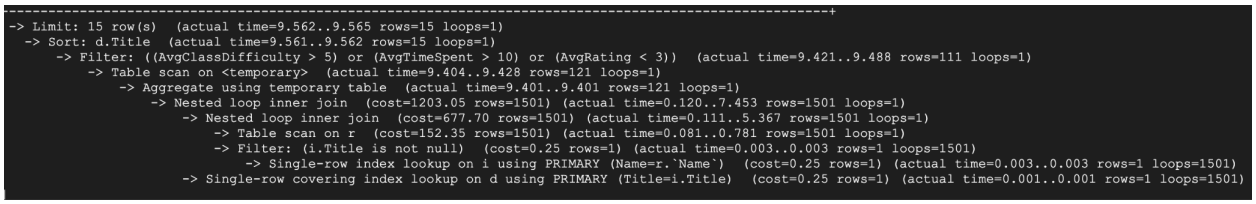
Before Indexing:



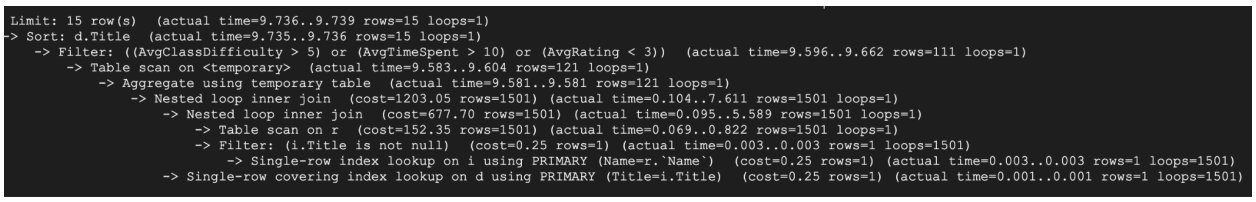
After Indexing on ClassDifficulty:



After Indexing on TimeSpent:



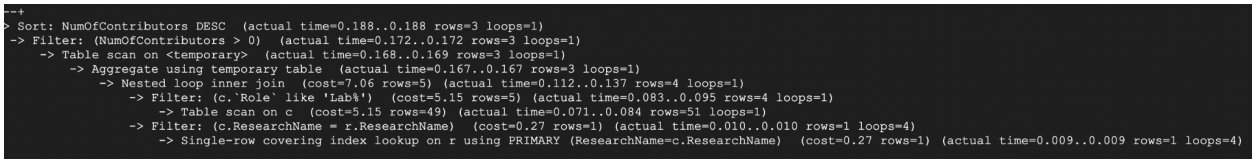
After Indexing on Rating:



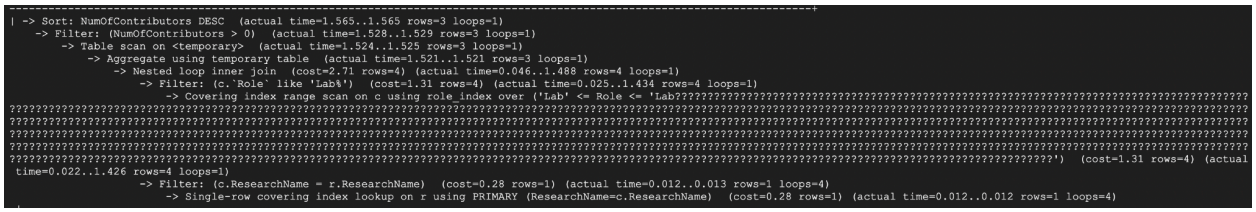
Justification: Indexing on ClassDifficulty, TimeSpent, and Rating did not change the cost of the query performance. This might be due to a few different reasons. All three of these attributes are in the HAVING clause of the advanced query, which will be calculated after GROUP BY. The more selective the HAVING clause is, the greater the optimization. The cost not changing much might imply that the HAVING attribute doesn't filter the GROUP BY aggregating much. Overall, indexing on these three attributes did not change much of the query's performance and efficiency in terms of cost.

Query 4 – Departments with Above-Average Class Difficulty Ratings

Before Indexing:



After Indexing Role:



Justification: For this query we are joining on our primary key so there are no additional attributes to index specifically for improving the JOIN performance. The cost appears to decrease after indexing, but the result of covering index range scan is several question marks. It implies that indexing on Role did not actually improve query performance, especially since many of the times increased as well. This is because the primary key already serves as a natural index, allowing fast retrieval of what's needed during JOIN.

Changes:

We completely redid our stage 2. We went back and fixed our UML diagram by making sure we accurately identified our entities and relationships. Furthermore, we fixed our relationship schema to fit our fixed UML diagram and determined the functional dependencies accordingly.